Apple Coding Drills: Concurrency and Rate Limiting (Python + Go)

Apple Coding Drills: Concurrency and Rate Limiting (Python + Go)

Run each drill in two passes:

1. Solve from memory in 20 to 30 minutes.
2. Compare against the reference and tighten edge-case handling.

Drill 1: Bounded Worker Pool

Problem: ### Flowchart

```
flowchart TD
    A[Input jobs and k] --> B[Init in out channels and res array]
    B --> C[Start k workers looping on in]
    C --> D[Send item i and v into in]
    D --> E[Worker computes sq = v times v]
    E --> F[Emit item i and sq to out]
    F --> G[Collector writes res at i equals sq]
    G --> H{Collected len jobs items}
    H -- No --> F
    H -- Yes --> I[Return res]
```

Implement a worker pool that processes integer jobs and returns processed results using exactly `k` workers.

```
from concurrent.futures import ThreadPoolExecutor

def worker_pool(jobs: list[int], k: int) -> list[int]:
    def work(x: int) -> int:
        return x * x

    with ThreadPoolExecutor(max_workers=k) as ex:
        return list(ex.map(work, jobs))
```

```go
func workerPool(jobs []int, k int) []int {
    type item struct {
        i int
        v int
    }
    in := make(chan item)
    out := make(chan item)
    res := make([]int, len(jobs))

    for w := 0; w < k; w++ {
        go func() {
            for it := range in {
                out <- item{i: it.i, v: it.v * it.v}
            }
        }()
    }

    go func() {
        for i, v := range jobs {
            in <- item{i: i, v: v}
        }
        close(in)
    }()

    for i := 0; i < len(jobs); i++ {
        it := <-out
        res[it.i] = it.v
    }
    return res
}
```

Drill 2: Token Bucket Rate Limiter

Problem: ### Flowchart

```
flowchart TD
    A[State rate cap tokens last] --> B[now equals current time]
    B --> C[elapsed equals now minus last]
    C --> D[tokens equals min cap and tokens plus elapsed times rate]
    D --> E[last equals now]
    E --> F{tokens gte 1}
    F -- Yes --> G[tokens minus equals 1 and return true]
    F -- No --> H[return false]
```

Allow up to `rate` tokens per second with `burst` capacity. Implement `allow()` that returns true or false.

```python
import time

class TokenBucket:
    def __init__(self, rate: float, burst: float):
        self.rate = rate
        self.cap = burst
        self.tokens = burst
        self.last = time.monotonic()

    def allow(self) -> bool:
        now = time.monotonic()
        self.tokens = min(self.cap, self.tokens + (now - self.last) * self.rate)
        self.last = now
        if self.tokens >= 1:
            self.tokens -= 1
            return True
        return False
```

```go
type TokenBucket struct {
    rate   float64
    cap    float64
    tokens float64
    last   time.Time
    mu     sync.Mutex
}

func NewTokenBucket(rate, burst float64) *TokenBucket {
    return &TokenBucket{rate: rate, cap: burst, tokens: burst, last: time.Now()}
}

func (b *TokenBucket) Allow() bool {
    b.mu.Lock()
    defer b.mu.Unlock()
    now := time.Now()
    elapsed := now.Sub(b.last).Seconds()
    b.tokens = math.Min(b.cap, b.tokens+elapsed*b.rate)
    b.last = now
    if b.tokens >= 1 {
        b.tokens -= 1
        return true
    }
    return false
}
```

Drill 3: Sliding Window Rate Limiter

Problem: ### Flowchart

```
flowchart TD
    A[State limit window and q timestamps] --> B[now equals current time]
    B --> C[Pop q front while now minus q0 gt window]
    C --> D{len q lt limit}
    D -- Yes --> E[Append now to q and return true]
    D -- No --> F[return false]
```

Allow at most `limit` requests per rolling `window_seconds`.

```python
import time
from collections import deque

class SlidingWindowLimiter:
    def __init__(self, limit: int, window_seconds: float):
        self.limit = limit
        self.window = window_seconds
        self.q = deque()

    def allow(self) -> bool:
        now = time.monotonic()
        while self.q and now - self.q[0] > self.window:
            self.q.popleft()
        if len(self.q) < self.limit:
            self.q.append(now)
            return True
        return False
```

```go
type SlidingWindowLimiter struct {
    limit  int
    window time.Duration
    q      []time.Time
    mu     sync.Mutex
}

func (l *SlidingWindowLimiter) Allow() bool {
    l.mu.Lock()
    defer l.mu.Unlock()
    now := time.Now()
    cutoff := now.Add(-l.window)
    i := 0
    for i < len(l.q) && l.q[i].Before(cutoff) {
        i++
    }
    l.q = l.q[i:]
```

```
    if len(l.q) < l.limit {
        l.q = append(l.q, now)
        return true
    }
    return false
}
```

Drill 4: Retry With Exponential Backoff

Problem: ### Flowchart

```
flowchart TD
    A[i equals 0] --> B{ i lt n }
    B -- No --> C[return false]
    B -- Yes --> D[ok equals fn call]
    D --> E{ok is true}
    E -- Yes --> F[return true]
    E -- No --> G{ i lt n minus 1 }
    G -- Yes --> H[sleep base times two power i]
    G -- No --> I[skip sleep]
    H --> J[i plus plus]
    I --> J
    J --> B
```

Retry a function up to `n` times with doubling delay and stop early on success.

```
import time
from typing import Callable

def retry(fn: Callable[[], bool], n: int, base: float = 0.05) -> bool:
    for i in range(n):
        if fn():
            return True
        if i < n - 1:
            time.sleep(base * (2 ** i))
    return False
```

```
func Retry(fn func() bool, n int, base time.Duration) bool {
    for i := 0; i < n; i++ {
        if fn() {
            return true
        }
        if i < n-1 {
            time.Sleep(base * time.Duration(1<<i))
        }
    }
    return false
}
```

Drill 5: Context-Cancelled Fan-Out

Problem: ### Flowchart

```
flowchart TD
    A[Input checks array] --> B[Create tasks for each check]
    B --> C[Await completed task result ok]
    C --> D{ok is false}
    D -- Yes --> E[Cancel remaining tasks]
    E --> F[Gather cancelled tasks]
    F --> G[return false]
    D -- No --> H{More tasks pending}
    H -- Yes --> C
    H -- No --> I[Gather done tasks and return true]
```

Run multiple checks in parallel and cancel all remaining work when any check fails.

```python
import asyncio

async def run_checks(checks):
    tasks = [asyncio.create_task(c()) for c in checks]
    try:
        for t in asyncio.as_completed(tasks):
            ok = await t
            if not ok:
                for p in tasks:
                    p.cancel()
                return False
        return True
    finally:
        await asyncio.gather(*tasks, return_exceptions=True)
```

```go
func RunChecks(ctx context.Context, checks []func(context.Context) error) error {
    g, ctx := errgroup.WithContext(ctx)
    for _, c := range checks {
        check := c
        g.Go(func() error { return check(ctx) })
    }
    return g.Wait()
}
```

Drill 6: Semaphore-Limited Concurrency

Problem: ### Flowchart

```
flowchart TD
    A[Input urls k fetch] --> B[Init semaphore sem with k permits]
    B --> C[for each url launch task one]
```

```
    C --> D[Acquire sem]
    D --> E[val equals await fetch url]
    E --> F[Release sem]
    F --> G[Store val in result index]
    G --> H{All urls completed}
    H -- No --> C
    H -- Yes --> I[return results]
```

Process a list of URLs with max `k` concurrent requests.

```python
import asyncio

async def fetch_all(urls, k, fetch):
    sem = asyncio.Semaphore(k)

    async def one(u):
        async with sem:
            return await fetch(u)

    return await asyncio.gather(*(one(u) for u in urls))
```

```go
func FetchAll(urls []string, k int, fetch func(string) (string, error)) ([]string, error) {
    sem := make(chan struct{}, k)
    res := make([]string, len(urls))
    g := new(errgroup.Group)
    for i, u := range urls {
        i, u := i, u
        g.Go(func() error {
            sem <- struct{}{}
            defer func() { <-sem }()
            v, err := fetch(u)
            if err != nil {
                return err
            }
            res[i] = v
            return nil
        })
    }
    return res, g.Wait()
}
```

Drill 7: Fixed Window Per-Key Limiter

Problem: ### Flowchart

```
flowchart TD
    A[Input key limit window] --> B[now epoch seconds]
    B --> C[bucket equals now div window]
```

```
    C --> D[counter key bucket plus equals 1]
    D --> E{counter lte limit}
    E -- Yes --> F[return true]
    E -- No --> G[return false]
```

Allow each tenant at most `limit` requests per `window`.

```python
import time

class FixedWindowLimiter:
    def __init__(self, limit: int, window_seconds: int):
        self.limit = limit
        self.window = window_seconds
        self.m = {}

    def allow(self, key: str) -> bool:
        now = int(time.time())
        bucket = now // self.window
        k = (key, bucket)
        self.m[k] = self.m.get(k, 0) + 1
        return self.m[k] <= self.limit
```

```go
type FixedWindowLimiter struct {
    limit  int
    window int64
    m      map[string]int
    mu     sync.Mutex
}

func (l *FixedWindowLimiter) Allow(key string) bool {
    l.mu.Lock()
    defer l.mu.Unlock()
    bucket := time.Now().Unix() / l.window
    k := fmt.Sprintf("%s:%d", key, bucket)
    l.m[k]++
    return l.m[k] <= l.limit
}
```

Drill 8: Leaky Bucket Queue Drain

Problem: ### Flowchart

```
flowchart TD
    A[State q and rate] --> B[interval equals one over rate]
    B --> C[item equals drain once]
    C --> D{item is none}
    D -- Yes --> E[sleep interval]
    D -- No --> F[handle item]
```

```
    F --> E
    E --> C
```

Accept bursts into a queue but drain at a steady rate.

```python
import threading
import time
from collections import deque

class LeakyBucket:
    def __init__(self, rate_per_sec: float):
        self.q = deque()
        self.rate = rate_per_sec
        self.lock = threading.Lock()

    def put(self, x):
        with self.lock:
            self.q.append(x)

    def drain_once(self):
        with self.lock:
            if not self.q:
                return None
            return self.q.popleft()

    def run(self, handle):
        interval = 1.0 / self.rate
        while True:
            item = self.drain_once()
            if item is not None:
                handle(item)
            time.sleep(interval)
```

```go
func RunLeakyBucket(ctx context.Context, in <-chan int, ratePerSec int, handle func(int)) {
    ticker := time.NewTicker(time.Second / time.Duration(ratePerSec))
    defer ticker.Stop()
    queue := make([]int, 0, 1024)
    for {
        select {
        case <-ctx.Done():
            return
        case v := <-in:
            queue = append(queue, v)
        case <-ticker.C:
            if len(queue) > 0 {
                v := queue[0]
                queue = queue[1:]
```

```
                handle(v)
            }
        }
    }
}
```

Drill 9: In-Flight Request De-Dup (Singleflight)

Problem: ### Flowchart

```
flowchart TD
    A[Input key and fn] --> B{key in inflight map}
    B -- Yes --> C[Wait event and return cached val err]
    B -- No --> D[Create event and result box]
    D --> E[Run fn once]
    E --> F[Set box val err]
    F --> G[Set event and delete inflight key]
    G --> H[Return box val err]
```

If concurrent callers ask for the same key, run work once and share result.

```
import threading

class SingleFlight:
    def __init__(self):
        self.mu = threading.Lock()
        self.inflight = {}

    def do(self, key, fn):
        with self.mu:
            if key in self.inflight:
                ev, box = self.inflight[key]
                wait = True
            else:
                ev, box = threading.Event(), {}
                self.inflight[key] = (ev, box)
                wait = False
        if wait:
            ev.wait()
            return box["v"], box.get("e")
        try:
            box["v"] = fn()
            box["e"] = None
        except Exception as e:
            box["v"] = None
            box["e"] = e
        finally:
```

```
            with self.mu:
                ev.set()
                del self.inflight[key]
        return box["v"], box["e"]
```

```
type call struct {
    wg  sync.WaitGroup
    val any
    err error
}

type SingleFlight struct {
    mu sync.Mutex
    m  map[string]*call
}

func (g *SingleFlight) Do(key string, fn func() (any, error)) (any, error) {
    g.mu.Lock()
    if g.m == nil {
        g.m = map[string]*call{}
    }
    if c, ok := g.m[key]; ok {
        g.mu.Unlock()
        c.wg.Wait()
        return c.val, c.err
    }
    c := &call{}
    c.wg.Add(1)
    g.m[key] = c
    g.mu.Unlock()

    c.val, c.err = fn()
    c.wg.Done()

    g.mu.Lock()
    delete(g.m, key)
    g.mu.Unlock()
    return c.val, c.err
}
```

Drill 10: Pipeline Gate Evaluator

Problem: ### Flowchart

```
flowchart TD
    A[Input tests security error budget approval] --> B[Init reasons empty]
    B --> C{tests false}
```

```
    C -- Yes --> D[append tests failed]
    C -- No --> E[next check]
    D --> E
    E --> F{security false}
    F -- Yes --> G[append security block]
    F -- No --> H[next check]
    G --> H
    H --> I{error budget ok false}
    I -- Yes --> J[append slo risk]
    I -- No --> K[next check]
    J --> K
    K --> L{needs approval and approval missing}
    L -- Yes --> M[append approval missing]
    L -- No --> N[evaluate output]
    M --> N
    N --> O{reasons empty}
    O -- Yes --> P[return promote and empty reasons]
    O -- No --> Q[return block and reasons]
```

Given checks (`tests`, `security`, `error_budget_ok`, `approval`), return `promote` or `block` with reasons.

```python
def evaluate_gate(inp: dict) -> tuple[str, list[str]]:
    reasons = []
    if not inp.get("tests"):
        reasons.append("tests_failed")
    if not inp.get("security"):
        reasons.append("security_block")
    if not inp.get("error_budget_ok"):
        reasons.append("slo_risk")
    if inp.get("needs_approval") and not inp.get("approval"):
        reasons.append("approval_missing")
    return ("promote", []) if not reasons else ("block", reasons)
```

```go
type GateInput struct {
    Tests         bool
    Security      bool
    ErrorBudgetOK bool
    NeedsApproval bool
    Approval      bool
}

func EvaluateGate(in GateInput) (string, []string) {
    reasons := []string{}
    if !in.Tests {
        reasons = append(reasons, "tests_failed")
    }
```

```
    if !in.Security {
        reasons = append(reasons, "security_block")
    }
    if !in.ErrorBudgetOK {
        reasons = append(reasons, "slo_risk")
    }
    if in.NeedsApproval && !in.Approval {
        reasons = append(reasons, "approval_missing")
    }
    if len(reasons) == 0 {
        return "promote", nil
    }
    return "block", reasons
}
```

Scoring Rubric

1. Correctness on edge cases.
2. Explicit concurrency bounds.
3. Cancellation and timeout handling.
4. Clear error classification.
5. Readable code under time pressure.

Weekly Drill Plan

1. Day 1: Drills 1 to 3 in Python and Go.
2. Day 2: Drills 4 to 6 in Python and Go.
3. Day 3: Drills 7 to 10 in Python and Go.
4. Day 4: Random four-drill no-notes mixed language round.
5. Day 5: Mock interview coding with verbal explanation.

Related notes:

1. Apple Panel Interview Prep Playbook (2026)
2. Apple Interview Story Bank and Incident Narratives (2026)